Graph Project: Communities detection

December 17, 2019

1 Introduction

Among the numerous clubs and association at N7 University, there is one in particular which raises our attention in this project. The old and venerable "karat7 club" composed this year of 34 persons is running into a crisis. Indeed the 2 most important managers of the club, Tristan the treasurer and Sophie the president, absolutely disagree on the organization and the content of the courses provided. Moreover, their relationship got a bit spicier since Sophie let Tristan made 90% of their duo graph project. Unfortunately, this is for sure that the club is going to split into 2 sub independant clubs... Your job in this project will be to predict which side each member of the club will choose knowing only the links between pairs of members who interacted outside of the club.

This problem is actually the classic Zachary's karate club problem, this is probably the most popular example of communiy structure. Apart from the little motivation scenario presented, community detection and the classification of individuals into groups is widely used into business models of social networks. To give few examples of standard applications:

- Present you relevant and targeted advertisements.
- Provide to companies dynamic information about communities for market studies.
- Or influence American presidential election ...

Of course a natural representation of individuals connections is graphs. Then to reach your objective which is about predicting the splitting of the

club before it happens, you will review 3 methods manipulating graph data. The two first ones ([2] and [3]) use edge scoring to iteratively extract communities. The last one [4] is a more modern approach for classification of individuals called graph convolutional neural network (a deep learning method). However, for this part you will only focus on the extension on graph of the convolution concept.

All the parts are independant. You will need to provide a little report where you answer the questions and CamL files where you will code your different functions.

2 Part 1: Girvan Newman algorithm

In this part you will classify in which team (Tristan team or Sophie team) each individual of the club will go thanks to the Girvan Newman algorithm. This algorithm determines the communities in people network graphs. To compute them, it uses the concept of edge betweenness defined for each edge as the number of shortest paths that run along it. I.e. considering all the shortest path between all pair combinations of vertices in the graph. The intuition behind is about considering that communities in a network are loosely connected by a few edges, which results in high betweenness on these particular edges. Hence, it is about removing them one by one to separate a strongly connected graph into i^{th} strongly connected components which let appear our i^{th} communities. The algorithm pseudo-code is described below in algorithm [1].

Algorithm 1 Girvan_Newman

while no edge left **or** desired number of communities unreached **do**
  Calculate betweenness of all edges.
  Remove the edge with the highest betweenness
  Calculate the number of strongly connected component (=communities)
end while

The karat7 club graph is represented in figure [1]. Each member of the club has an ID number between 1 and 34 and he/she is labeled according to the real choice he/she will make during the splitting. The 1^{st} vertex is Tristan and the 34^{th} is Sophie. Each edge carry a score counter initialized to 0 and representing their edge betweenness. You may need to display your graph

for debugging or question answering, please refer to the part [5.2] for more information. The karat7 club graph data is located in the module `Zachgraph` of the source.

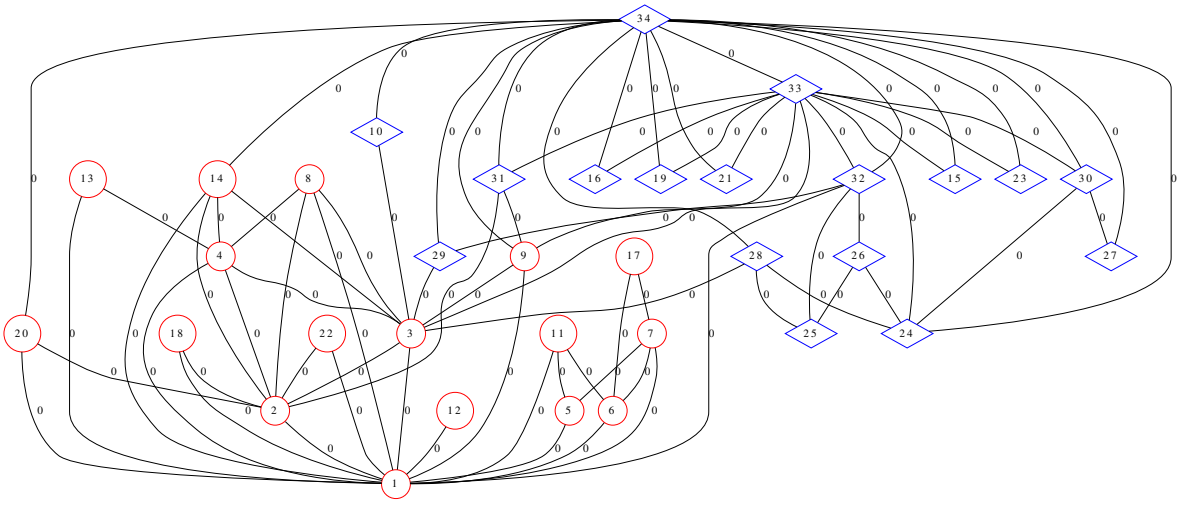

Figure 1: The people network graph representation of the karat7 club.

The graph you will manipulate are created through the `Zach` module presents in the sources. Most of the functions of your standard `Graph.Pack.Graph` module used in TP are compatible and callable with `Zach` (so don't hesitate to **read the doc**). Neverthess we give a summary of the main additional functions you may use in annex.

Question 2.1

Manually apply the Girvan Newman algorithm on graph `g1` defined in the module `Samplegraph` to separate $k = 2$ communities. Display each iterations with the computed betweenness on the edges in your report.

Indication

To don't bother draw a graph in your report, you are encouraged to use the function `setscore` and `remove_edge_e` of the `Zach` module to mark and remove edges. Then plot the graph as explained in part [5.2].

Code 2.1 Code the function betweenness_score

Compute the betweenness of all edges. The betweeness of an edge is the number of shortest path between all the vertices of the graph which pass through it. Write the function `betweenness_score graph` part of the module `girvan.ml` where `graph` is a given people network graph.

Indication

- You **don't** need to **recode a shortest path** yourself. Read the doc ... (and the annex)
- There are functions to manipulate the score of an edge, they are part of the additional functions of `Zach` described in annex.
- **To simplify the coding** we highly suggest you to have a greedy approach of the form "for each vertex compute the shortest path with all the other vertex of the graph" which actually computes two times more each shortest path but don't change the algorithm result at the end.

Code 2.2 Code the function girvan_newman

Compute the n communities of a given graph. Write the function `girvan_newman graph n_com` part of the module `girvan.ml` where `graph` is a given people network graph and `n_com` is the number of communities to extract.

Indication

- You **don't** need to **recode a strongly connected components detection** yourself.
- The computation is done in place. You will remove edges of the graph you are working on without doing any copies.

Tips: Apply `girvan_newman` on the graph `g1`. You should get the same separation as for question 2.1. The betweenness on the edges should be the

double of what you computed manually. It could be a good idea to see if your algorithm works on graph `g1` and `g2` too.

Question 2.2

According to the results you obtained on the karat7 graph, what do you observe and conclude on its efficiency and limitation(s).

Question 2.3

By observing the graph `g2` of the module `Samplegraph` tell what improvement in algorithm 1 can speed up a bit the splitting.

Indication

It concerns the computation of the betweenness.

3 Part 2: Neighborhood Overlap (NOVER) algorithm

In part [2] you created an algorithm able to separate a graph in k communities, k being known. This is actually a really logical approach for handling the karat7 club problem where we know that there are exactly two teams, so two communities to distinguish : Tristan's one and Sophie's one. However, there are many applications where you don't have any information about the number of communities in the graph, and part of the job is to determine this number. We will particularly focus on the neighborhood overlap approach supposed to tackle this issue in this part.

As for the betweenness in the Girvan Newman algorithm [1] we will define a score on each edges: the neighborhood overlapping or NOVER score. This value measure how much the two vertices of an edge carry the same neighborhood. If they do, they are probably in the same community, if not, they are probably not.

$$S_{nover}(v_1, v_2) = \frac{\text{Number of nodes who are neighbors of both } v_1 \text{ and } v_2}{\text{Number of nodes who are neighbors of at least } v_1 \text{ or } v_2}$$

Note that one should not count neither v_1 or v_2 as part of the neighbors in the denominator.

Then the algorithm will be about to iteratively remove the edges with the smallest nover score. The communities will appear after a certain number of edges removals as for algorithm [1]. However, there is a subtlety here: you don't know the number of community, so you don't know when to stop. Hence you need to introduce another parameter which determines when you reach the most probable community separation. This parameter is the modularity of a graph which indicates the strength of a division into communities of a graph. Network with high modularity have dense connections between the nodes within communities.

$$Mod = \sum_{\text{Communities}} \sum_{(i,j)\in \text{Vertices}} A_{i,j} - \frac{d_i d_j}{2m}$$

Where A is the adjacency matrix, so if it exists an edge between v_i and v_j $A_{i,j} = 1$, $A_{i,j} = 0$ otherwise. d_i is the degree of v_i and m the initial number of edge in the graph.

Algorithm 2 NOVER

while no edge left **or** thresh community number exceeded **do**
  Calculate nover score of all edges.
  Remove **all** the edges with the smallest nover score.
  If the graph get disconnected compute the modularity score of the new division.
end while
The set of communities with the highest modularity is the optimal community partition of the graph.

As for part 2 you need to use the score counter of the edge. We invite you to display [5.2] the graph with the computed nover score on the edges for debug or question answering.

Question 3.1

Manually apply the NOVER algorithm on graph `g3` defined in the module `Samplegraph`. Display each iterations with the computed nover score on the edges in your report plus the details of the modularity

computation.

Indication

- To don't bother draw a graph in your report for each iteration, you are encouraged to use the function `setscore` and `remove_edge_e` of the `Zach` module to mark and remove edges. Then plot the graph as explained in part [5.2].
- For convenience you will **just do 3 iterations of the loop**. Stop when there is no edge left may be a bit long ...

Code 3.1 Code the function nover_score

Compute the nover score of all edges. The nover score has been defined earlier in the part. You will need to count the number of common neighbors of two vertices and the number of vertices which are neighbor of at least one of the two vertices. Write the function `nover_score graph` part of the module `nover.ml` where `graph` is a given people network graph.

Algorithm 3 step_nover pseudo-code

while no new community separation **do**
  Calculate the nover score of all edges.
  Remove **all** the edges with the smallest nover score.
end while

Code 3.2 Code the function step_nover

For convenience you will not code the algorithm as written in algorithm [2]. You will need to code the step_nover described in algorithm [3] which stops when it has reached a community separation. Write the function `step_nover graph` part of the module `nover.ml` where `graph` is a given people network graph.

Indication

As the edge removing is permanent, you can apply successively `step_nover` on the graph to observe the community separation step by step.

Question 3.2

Apply `step_nover` few time on the karat7 graph (3∼4 times). Display the graphs obtained at each application of the function in your report. Compare your result with the ones obtained using the Girvan Newman algorithm.

Code 3.3 Code the function modularity

Compute the modularity of the graph as defined previously in the part. Write the function `modularity graph communities` part of the module `nover.ml` where `graph` is a given people network graph and `communities` is the list of communities in the graph.

Indication

- As the modularity is only computed on the original graph, you need to be carful to **provide** to the function **the original graph** at test time. `step_nover` removes edges from the graph and hence changes it !
- Enter the communities list manually.

Tips: Apply `step_nover` 3 times on the graph `g3`. Use your `modularity` function to compute the modularity for each iteration. You should get the same results as for question 3.1. It could be a good idea to see if your algorithm works on graph `g2` and `g1`.

Indication

- You may need to proceed in two steps, first apply `step_nover` 3 times and observe the resulted communities. And then take back the original graph and hard write the communities as the arguments of your `modularity` function.

Question 3.3

Apply `step_nover` few times on the `karat7` graph. Use your `modularity` function to compute the modularity for each iteration and display in your report the results. Conclude on the best separation (number of communities) proposed by the NOVER algorithm. Does it look correct ?

Question 3.4 BONUS: These questions may require a bit of imagination

According to the result obtained using the NOVER approach on the karat7 club, which person may totally stop the karate after the separation of the club ?
Do you observe a limitation using the NOVER approach on the karat7 graph ? You can focus on a particular vertex to explain.

4 Part 3: Graph convolution

In order to process the gigantic graphs from the social networks combine with complex information gathered on each individuals, new technics has been developped recently. One of the most trending one is probably the graph convolutional neural network which has shown incredible results on classification of individuals in communities, ie. the problem we are interested in in this project.

In this part we will not ask you to manipulate neural networks, that is why you will unfortunately not be able to run any classification on the karat7 graph. We are actually interested in the graph convolution concept which is the baseline of this technic. A convolution operation is normally defined on

euclidian object like matrices as described in figure [2]. Visually this is about sliding a window over a matrix while applying a kernel on it.

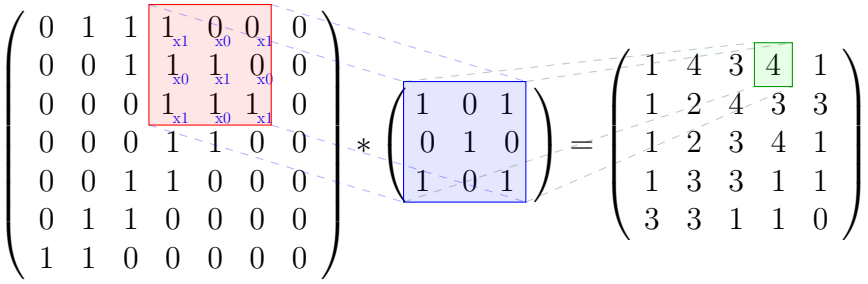

Figure 2: Illustration of convolution on matrix

It can be written that way:

$$c(x,y) = w * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t)$$

where f is the original matrix and w the kernel. The strength of this tool and the reason why it is widely used among many applications including neural networks is the fact that it propagates the neighbouring information. Moreover, particularly in people networks, one of the most relevant information about somebody is carried by his/her close neighbours. However graphs are not euclidian objects, hence we can not simply translate the convolution previously defined on this kind of data.

That is why we propose here to define and implement a first order convolution on graph which on each vertex computes the mean of the parameters of its neighbours. The main difference compare to the euclidian convolution is the complexity to define the kernel since the number of neighbours of each vertex varies from vertex to vertex. We will not enter into details for this specific issue and just consider that the kernel is the averaging operator.

$$\forall v \in g,\ c(v) = \frac{1}{|\mathcal{N}(v)| + 1} * \Big(\sum_{u \in \mathcal{N}(v)} h_u + h_v \Big)$$

where h_v is the parameters of the vertex v (basically a vector of $\mathcal{R}^p$), $\mathcal{N}(v)$ is the neighbours of v). Figure 3 represents a more visual way to approach this concept.

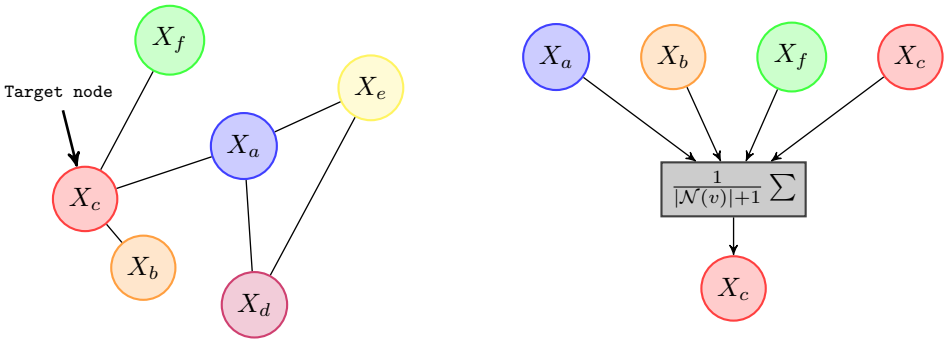

Figure 3: Illustration of a graph convolution on a vertex

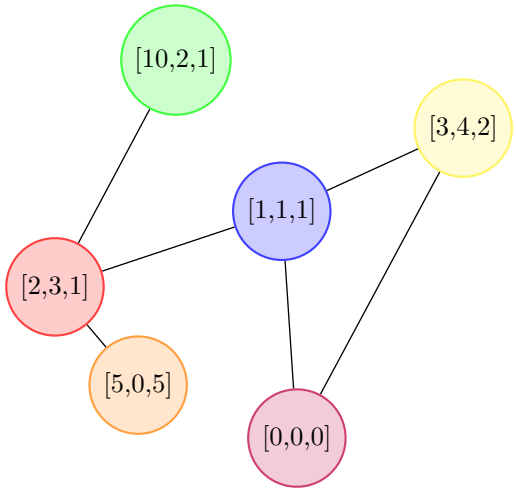

Figure 4: Graph carrying parameters

Question 4.1 Understand the convolution

Apply manually the convolution defined in this part on the graph in figure [4] (= graph `g4` in `Samplegraph`).

Code 4.1 Code the function convolution

Write the function `convolution graph ver` part of the module `convolution.ml` where `graph` is a given people network graph and `ver` is a vertex which computes the new parameters of `ver`.

Code 4.2 Code the function print_param

Write the function `print_param graph` part of the module `convolution.ml` where `graph` is a given people network graph which prints the result of the convolution for all the vertices of the `graph`.

Indication

We **don't** ask you to **store the parameters** computed by the convolution in the graph data structure.

Question 4.2

Why big social network with the data they own about you can not only settle for algorithm of type Girvan Newman or NOVER ?

5 Annex

5.1 Instructions

This is a **duo** project.

Each team have to provide the `.ml` files implementing the interface `.mli`:

- `girvan.ml`
- `nover.ml`
- `convolution.ml`

Plus a **short report** (in english or in french) answering the questions of the subject and displaying your community graphs as asked in the questions.

Part of the evaluation is about using a functional programming paradigm. So no `while` loop, no `for` loop and no references.

Deadline : 24 January 2019 at 23h59.

5.2 Display a graph

In order to display a full graph in a PDF file, you can call the function `Graph.Pack.Graph.dot_output your_graph your_file` in an ocaml script. Then in your terminal type `dot your_file.dot -Tps -o your_file.ps`. Finally, to visualize, open `your_file.ps` with your favourite PDF viewer.

5.3 Compile the project

To compile your files (the modules `girvan.ml nover.ml convolution.ml` plus your `tests.ml` files) with the `Zach` graph module structure and the different graph samples (`zachgraph.ml` and `samplegraph.ml`) you can type:

```
ocamlfind ocamlopt -o exec -linkpkg -package ocamlgraph zach.ml
 zachgraph.ml samplegraph.ml girvan.ml nover.ml convolution.ml
                          your_tests.ml
```

5.4 Module functions

You will work with the `Zach` module which is just a module built over your normal ocamlgraph tools. Most of the functions described in the ocamlgraph doc are usable (ocamlgraph.lri.fr/doc/Pack.html) with `Zach`: iter, fold, map, etc. However a `Zach` graph has few more features:

- Each vertex carry a number (its name), a label (in which team it is), and a parameters list (list of float).
- Each edge carry a score and this is the value printed on the edge when using the display option.
- `createv : int*int*(float list) -> Zach.Zach.V.t`

  Create a vertex with (number, label, parameters list)
- `createe : Zach.Zach.V.t -> int -> Zach.Zach.V.t -> Zach.Zach.E.t`

  Create an edge with 2 vertices plus a number (its name).

- `getparamv : Zach.Zach.V.t -> float list`
  Get the list of parameters of an edge.
- `getscore : Zach.Zach.E.t -> float`
  Get the score of an edge.
- `setscore : float -> Zach.Zach.E.t -> unit`
  Put the score of an edge at a certain value.
- `incrscore : float -> Zach.Zach.E.t -> unit`
  Increment the score of an edge with a certain value.
- `shortest_path : Zach.Zach.t -> Zach.Zach.V.t -> Zach.Zach.V.t -> (Zach.Zach.E.t list)*int`
  Compute the shortest path between two vertices.
- `components : Zach.Zach.t -> Zach.Zach.V.t list list`
  Lists of the vertices of the different strongly connected components.
- `getsrc : Zach.Zach.E.t -> Zach.Zach.V.t`
  Source of the edge.
- `getdst : Zach.Zach.E.t -> Zach.Zach.V.t`
  Destination of the edge.
- `getnumberv : Zach.Zach.V.t -> int`
  Number of the vertex.
- `getnumbere : Zach.Zach.E.t -> int`
  Number of the edge.
- `choosev : Zach.Zach.t -> Zach.Zach.V.t`
  Returns a vertex from the graph.
- `choosee : Zach.Zach.t -> Zach.Zach.E.t`
  Returns an edge from the graph.